A STAVE for PANDA: Systems Trace Analysis Visualization (Entire)

Tom Boning

May 18, 2015

1 Abstract

The Platform for Architecture Neutral Dynamic Analyis (PANDA) creates easy mechanism to collect very low level data about a system recording created in it, but lacks higher level information overviews. Systems Trace Analysis Visualization (Entire) creates a fast high level view of approximated information flows between processes in the system trace. This allows reverse engineers using PANDA a way to target certain processes for further study without needing slow and heavy weight system analysis.

2 Overview

The Platform for Architecture Neutral Dynamic Analyis (PANDA) is a reverse engineering tool developed by Lincoln Lab, Georgia Tech and Northeastern University. This tool is an open source, whole systems emulator built on top of QEMU that allows repeatable reverse engineering of dynamic programs through a record and replay mechanism. These recordings capture all inputs (DMA, interupts, network, etc) for the emulated system in a repeatable manner[1]. PANDA also supports a plugin system that allows call back functions to be executed when conditions are met, such as on system calls. These are composable, allowing high complexity analysis to be built easily.

PANDA allows analysts to gain insight into the program they are reverse engineering through the record and replay system. It allows high fidelity replay of an entire operating system, such as Windows, where the processes are interleaved in the exact same way each replay, as well as being in the same memory locations. This enables iterative reverse engineering on dynamic code, as the analyst can employ different plugins and build up knowledge about what the program is doing over time and at various levels of abstraction. This iterative approach provdes a benefit, especially when code may execute in unpredictable manners.

Although PANDA offers a very fine grain view into the execution of a program, it has less to offer for quickly gaining whole system understanding, which is especially useful at the beginning of a reverse engineering task. This leads to analysts writing similar code over and over when they start working with a replay. General information, such as what processes exist, how they create and interact with one another, and the information flow through the processes, is especially useful to a reverse engineering analyst at the start of a task. What is needed is an easy way to visualize a high level abstraction of the whole system trace that can be run cheaply and routinely, allowing a foundation for deeper levels of reverse engineering knowledge to be collected. In terms of cheapness, we seek to have a less than five times slowdown for a replay, which will be substantially faster than dynamic taint analysis. To satisfy this, Systems Trace Analysis Visualization (Entire) (STAVE), allows fast systems analysis presented in an easy to understand visualization. Although PANDA is architecture neutral, the initial scope of STAVE is Windows 7, 32 bit.

What STAVE does is track the flows of information through prominent Windows system calls. This allows it to estimate information flows and communication. The targeted system calls represent different mechanisms that processes use to communicate with one another. The currently instrumented system calls can be grouped into a few catagories: File Reading and Writing, Windows Registry Queries, Shared Memory Sections, and Windows Asynchronous Local Procedure Calls, and direct virtual memory reading and writing. Together, these systems give a good high level view of interprocess communication. Once collected, the data is output through the Pandalog system.

STAVE's goal is not to exhaustively collect all data about information flows between processes. Instead, it aims to provide an overapproximation of flows such that further examination can determine exactly how to flow behaves. This approximation is an overestimate for the implemented inter process communication mechanisms, as in case of doubt about the direction or actual existance of a flow, we mark it as a flow. We're willing to accept this overestimate as it provides an acceptable tradeoff for speed performance, and still provides useful information, as a flow that doesn't really exist can be looked at and discarded later. However, missing an existing flow would be problematic as it wouldn't allow an analyst to know what to look for. We also do not aim to be exhaustive about the various 'clever' ways that processes could signal each other. For instance, processes could infer state from looking at timing information from how long operations take to run. Though processes, especially malware, may wish to use such sneaky mechanisms, there are simply to many to catalog in a starting plugin.

Once data is collected about the interprocess communication mechanisms, we match different in and out flows together depending on the mechanism. For each syscall, we create a flow in or out of the process. For instance, reading data is an in-flow of information to that process. Other flows indicate bi-directional information flow, such as shared memory. Often for these, we cannot estimate at a finer grain level of detail without heavier weight analysis, such as taint based approaches. There are other times when we can see both sides of the flow, such as when a process maps a shared memory section into the virtual adress space of another process. After data collection and analysis, STAVE displays the data in a simple matrix of sender process to receiver processes. This data display is extendable easily as it is similar to csv, allowing graphical visualizations to be made on it.

Although processes may be sneaky about transfering information, STAVE does not attempt to provide a complete detection system for the difficult to track flows. Instead the aim is to provide a useful starting guide that can be used to locate places for further examination.

3 Data Extraction

The first part of STAVE is collecting data about various windows system calls so that we can collect data about flows between them. To do this, I extended the existing "win7proc" PANDA plugin. This plugin uses PANDA to track the Windows 7 system calls, and is based off of the more generic "syscalls2" plugin, which requires an operating system profile and then presents call back hooks based off of those system calls. The File System system calls already existed in win7proc and the Registry System calls already had most of their instrumentation. However, the shared memory sections system calls were not present and required implementation in win7proc, and the ALPC system calls and Virtual Memeory system calls also required implementation. Once data is collected, this data is output through Pandalog, a preexisting unified protocal buffer based logging system. This logging system is used by many plugins and provides a unified data source for plugins to output to a file.

This system of data extraction is faster than other finer grain analyses, such as Dynamic Taint Analysis. Dynamic Taint Analsis tracks information flows at the data level, allowing data to be tagged and it's progress through the system tracked. A simple analogy for this style of system would be adding dye to a system of water ways and seeing where that dye goes. Despite the power of this analysis, it requires overhead on every memory read or write, which is a quite costly performance hit. Dynamic Taint analysis requires a slowdown of 30-40x, while STAVE only requires a slowdown of 3-4x.

3.1 PANDA data collection

Instrumentation required some reverse engineering knowledge. Although PANDA gives hooks into the arguements of the system calls, their return values and the virtual address space of the process, it does not have high level tools for pulling data structures out of this memory. This is because PANDA exposes the system trace recording memory through the *panda_virtual_memory_rw* function, which takes in a the current CPUState, the input pointer as a 32 bit value, an output array and how much memory to read. This can also be used to set memory values, but that is not wanted when we simply want to introspect into a system recording. The main issue with this, is the complete lack of a type system, which is compounded by not knowing what the correct types are anyways. Additionally, just knowing the types does not give the full context of what that type semantically means. For instance, Handle objects are opaque to the user program, but we need to sematically understand what each means to extract useful information. Thankfully, existing reverse engineered knowledge exists, such as the data structures used by the Windows operating system, from programs such as the Volatility Framework[5]. Of course, it is still necessary to figure out the exact correspondance between data types and the pointers, as well as actually pull out useful information.

For instance, to read a Windows File Handle, even using a function that reads a process handle table, and one that sematically understands where the file name lives inside the file handle object, the following is required:

```
uint32_t handle;
panda_virtual_memory_rw(env, FileHandle, (uint8_t *)&handle, 4, false);
if (handle) {
  HandleObject *ho = get_handle_object(env, get_current_proc(env), handle);
  printf("    File Handle: %d: %s\n", ho->objType, get_handle_object_name(env, ho));
}
```

Many of the Windows System Calls are documented on Microsoft's documentation website, https://msdn.microsoft.com[3], though some are not. The naming scheme of a system call for Windows is generally of the form NT*OperationType*, where *Operation* is some operation performed, and *Type* is a generalized class of system calls, such as File or Section. For system calls that microsoft does not document, some sites exist that try to document them. If these system calls are called from the kernel instead of userspace, they are prefixed with ZW instead of NT. These sites house information that has been reverse engineered from Windows already, such as http://undocumented.ntinternals.net/. For instance, the Asynchronous Local Procedure Call system calls are not documented officially, as they are not exposed to developers by the Windows Kernel.

Note that we are ignoring the validity of system calls. We do not check the return values to ensure that the correct parameters were passed or that the security context is valid. This menas that we assume success when there may not have been, an example of the over-approximation we aim for.

3.2 Files

The file system calls output information based on the file name and the process. In these system calls, a File Handle is required that contains the file name, an easy way to uniquely identify files.

Windows uses File system calls for a variety of different I/O mechanisms[4], including directory modification and file streams. By hooking into these system calls, we can observe a variety of data flows between processes through the file system or other IO devices.

3.2.1 File System Calls

- **NTCreateFile**: Creates or opens a file, returning the File Handle to the caller of the system call.
- **NTReadFile**: Reads data from a file using an existing already open File Handle.
- **NTDeleteFile**: Deletes a file, closing the File Handle.
- **NTWriteFile**: Writes data to a file, using an existing already open File Handle.

3.3 Shared Memory Sections

The shared memory section system calls are used by Windows to handle several different types of shared memory and memory mapping. This includes mapping file contents into memory, for code execution for instance. Important system calls in this category are *NTCreateSection* and *NTOpenSection*, which return a section handle, which creates the shared memory section. Although some sections are "named" for access by other process, the most common style of sections are "anonymous" which do not provide a unique string based name. For this reason, we track the memory location of a section pointed to by a section handle as a unique identifier for the section, if a section name or file name does not exist. This may not correctly distinguish different Sections, such as a process closing the section handle and opening a new section handle in the same reallocated memory space. However, this over appoximates the sections system, which is acceptable. This works only because section objects cannot be passed easily between different processes, as doing so requires a name, which we can track as a way to identify that section.

3.3.1 Section System Calls

- **NTCreateSection**: Creates a new section, returning the Section Handle to the caller.

- **NTOpenSection**: Opens an existing section, returning the Section Handle to the caller. This requires a way to reference the existing section, such as a name or file name.
- **NTMapViewOfSection**: Maps an existing section, refereced by the Section Handle into a process's virtual address space. This can allow a process to map the section into another process's virtual address space if the Process Handle input is different than the current process.

3.4 Windows Registry

The Windows registry provides a hierarchical database that stores configuration data for both the Windows operating system and local user programs. The windows registry uses a set of several mappings to get from *key* to *value* to data. However, this system also allows the 'value' to instead be a *subkey*, allowing recursive nesting of key value stores. In order to uniquely identify the data, STAVE tracks the hierarchy through concatenating the parent key and the value name, which allows easy recursive tracking, in a similar way to a file paths. This is what the windows documentation does for subkeys, which is a consistent and simple system. There are a few versions of each system call, including the *NTFooKeyEx NTFooKeyTransacted* versions, which take in extra information about what to do, and allow creating transactions of registry system calls.

3.4.1 Registry System Calls

- **NTCreateKey**: Creates a new key or subkey, returning a Key Handle to be used for future system calls.
- **NTOpenKey**: Opens an existing key or subkey, returning a Key Handle to be used for future system calls.
- **NTDeleteKey**: Deletes an existing key or subkey, closing the Handle.
- **NTQueryKey**: Returns information about a key or subkey, including the class of the key and the number and size of subkeys.
- **NTQueryValueKey**: Returns a value entry's data.

- **NTDeleteValueKey**: Deletes a value entry.
- **NTEnumerateKey**: Provides information about subkeys in a key or subkey.
- **NTEnumerateValueKey**: Provides information about value entries, depending on an input value, in a key or subkey. This includes the value entry's data as well as the value entry name.
- **NTSetValueKey**: Sets a value entry's data to the input, or creates a new value entry for that value name.

3.5 Asynchronous Local Procedure Calls

The Windows kernel also uses a Asynchronous Local Procedure Call (ALPC) mechanism to do interprocess communication in a client server model. These system calls aren't exposed to user programs directly, but are instead layered in other system calls and are an undocumented system of the closed source windows operating system. This client server model does provide an easy way to tell what processes are communicating, as the client and the server processes are held by the port structures pointed to by a joint communication information structure. This allows us to get the two processes at the end of a successful port connection set up easily. Since they are undocumented, the best partial documentation can be found in the http://undocumented.ntinternals.com site

3.5.1 ALPC System Calls

- **NTCreatePort**: Creates a new ALPC Port, returning a Port Handle.
- **NTConnectPort**: Connects to an existing ALPC Port specficied by a Port Name.
- **NTListenPort**: Initializes a created port to listen for incoming connection requests
- **NTAcceptConnectPort**: gives the server a Port Handle for the client.
- **NtCompleteConnectPort**: Completes a port connection request by the sever.
- **NtRequestPort**: Sends a port message.
- **NtRequestWaitReplyPort**: Sends a port message, waiting on the reply.

- **NtReplyPort**: Replies to a port message.
- **NtReplyWaitReplyPort**: Responds to a RequestWaitReply, waiting for the reply.
- **NtReplyWaitReceivePort**: Responds to a reply, waiting for the next request. Typically used by the sever.
- **NtImpersonateClientOfPort**: Used to get the client's security context.

3.6 Virtual Memory Operations

Windows 7 allows reading and writing other process's virtual memory through the aptly named *NTReadVirtualMemory*, and *NTWriteVirtualMemory* system calls, which are given to user applications through a wrapper. Both of these provide full flows of information, from the caller process to a targeted process. Although there are several Virtual Memory system calls in windows, such as allocators and deallocators, the pair we care about are the system calls for reading and writing memory.

3.6.1 VM System Calls

- **NTReadVirtualMemory**: Reads memory starting at a virtual address space base adress into a buffer for the caller. The targeted process is identified by a Process Handle.
- **NTWriteVirtualMemory**: Writes memory from a buffer to the array starting at a virtual address space base adress. The targeted process is identified by a Process Handle.

3.7 Pandalog

The Pandalog system is an existing protocal buffer based logging systems for use by PANDA plugins. In particular, Pandalog is designed to be

- Fast to read and write
- Small log size
- Easy to add to a plugin
- Easy to write code that reads the log

- Useable from any C or C++ panda plugin

according to the design documents for it[6]. The “win7proc” plugin already used pandalog for existing plugins, but required extension for the section, virtual memory, and ALPC system call data.

3.8 Data Extration Performance

STAVE requires a slowdown of about 3-4x. This cost is associated with the slowdown due to hooking into system calls, halting the progression of a replay while analysis is done. It’s not currently feasible to parrallelize this with the replay, due to needing to read information out of memory from the replay. However, this is vastly faster than Dynamic Taint Analysis. Though Dynamic Taint Analysis depends on what input information is tagged and how to program propagates this data, it ussually requires a 30-40x slowdown[2]. Since we are much faster than this, we have acheived cheapness. In practice, a normal replay can take upwards of 5 minutes to run without plugins. A 20 minute replay for our analysis is much faster than a 200 minute replay in practical terms.

4 Data Analysis

After output to a pandalog file, the file needs to be parsed and analyzed for flows. Some flows are a complete flow from sender to receiver, but most are ‘half’ flows from or to an intermediary, such as the registry or a file. We store the inflows and outflows, and then compare each against each other. Inflows and Outflows may have multiple senders and receivers for a given flow, if a file has multiple process read and write to it for instance. If the process is the same as the one that sent the flow, we handle this the same as for another process. For full-flows, we store the full flows and add them to the matched flows for output.

The current matching system is naive, and tries to match each flow against each other flow. More sophisticated approaches could be used, such as tracking the different types of flows in order to speed the matching. In practice, this is unneccessary, as the number of flows to check small enough. Currently even examining pandalog files from long replays takes under a second to process, which is insignifigant compared to the replay timing cost.

This flow matching system is a huge overapproximation. Though we look at possible flows, we assume that there could be a flow where there may not be at all. For instance, if a file reads the start of a log right after another file appended to the end of it, this would be marked as a flow even though the data from the first process is not going into the second. This line of reasoning applies to registry, section and vm operations to a degree. Due to the need to set up a connection for ALPC, it is unlikely that processes take information from request messages and discard it without looking at it.

Another issue with the current system is that it does not take time into account. For instance, if Process A reads a file 'HelloWorld.exe' before Process B writes to it, this system will still associate a flow between the processes. This is acceptable, as in practice these overestimates do not happen frequently. Fixing it wouldn't be that difficult. Another issue currently is that we don't track file uniqueness well, to ensure that the file names are actually pointing to the correct file. For instance, if a file is deleted and recreated, we don't differentiate it currently. Again, this is an overestimate.

4.1 Files

Files are quintesential example of a partial flow system. Each file system call is indicative of a partial flow: *NTReadFile*, *NTOpenFile*, and *NTCreateFile* show in flows towards the process the opened the file. *NTWriteFile* shows an outflow. Once we have these partial flows, we can match them against each other based on the file name, as it uniquely identifies a flow.

4.2 Shared Memory Sections

Memory Sections are harder to determine flows from, as we cannot see flows being sent and received using them. To do this, we would need to watch memory operations, as memory does not require system calls to write and read from. Introspection on memory operations requires much heavier weight analysis systems which are prohibitively slow, such as taint. However, we can overestimate that there is two way communication for shared memory between processes with the same sections. This means that each of the *NTCreateSection* and *NTOpenSection* syscalls imply both an in and an out flow, as multiple processes may open and examine the same shared memory section. In addition, section syscalls may imply a file in flow, as they may optionally load a file to execute. The *NTMapViewOfSection* can be used to map a memory section into another processes virtual address space, which is a full flow between the two processes.

Another issue with memory sections is their anonymity. If a process opens a section that is unnamed, which prevents other processes from being about to get a handle to it, we may overapproximate the number of sections due to the lack of a ‘proper’ unique identifier, such as a name or UID. Since we use the memory location of the section handle in the process’ address space, we may have false positives, where two different sections are thought to be the same. If memory sections use a name or file name, we don’t have this problem.

4.3 Windows Registry

The Windows Registry is another half flow system, based around querying values and setting values. The *NTQueryKeyValue* reads data out of the registry, an inflow into the process. The *NTSetKeyValue* system call sets the registry value, an outflow. In addition, there are *NTQueryKeyValueEx* and *NTSetKeyValueEx* versions of these system calls, which are transaction versions. For our purposes, both types have the same style of flow.

4.4 Asynchronous Local Procedure Calls

ALPC system calls also are by their nature full flows between a client process and a server process, as they use a reply based system. Upon initialization a name is associated with a ALPC server, but we don’t need to use this for identification. Instead, we directly figure out what processes talking, and create a full flow based on that between the processes.

4.5 Virtual Memory Operations

NTReadVirtualMemory and *NTWriteVirtualMemory* are easy to separate into flow directions in or out of a process. Since they are full flows, we can add the flows to the process that is the ‘sender’ of information.

5 Visualization

The goal of STAVE was to provide a way to visualize process flows. Though the current visualization system is text based, it could be extended to a graphical system that would allow deeper inspection into flows.

5.1 Current

The current visualization is based on a adjency graph style display. Each process that is the ‘sender’ of information has a list of ‘receiver’ processes, with a count of each communication type used. This system is equivilant to creating a directed graph of information flows, with processes as nodes. In addition to the list of edges representing a flow existing, the count of the information flow is presented. This allows the user of STAVE to see ‘noisy’ processes that communicate highly and get a sense of how common certain forms of communication are in relative terms. However, this does not track the quantity of bytes passing between the processes, just the number of system calls, which are relative to the type and style of the program. Note that if a process pulls in information from another, that process will be displayed as the ‘sender’ in this system even if it does not actively ‘push’ the information.

5.2 Extensions

A useful extension of STAVE would be to display the actual information in a coded graph. For istance, a color could be assigned to each type of information flow, with directed edges and the width of edges displaying the commonness of an information flow. An interactive version of this would also allow you to delete processes and edges the user doesn’t care about, such as various standard operating systems components communicating.

6 Examples

To show how STAVE can be used to gain a starting insight to a program trace recording, we will show the effectiveness of it on several examples.

6.1 Kaspersky Anti-Virus

The first example that STAVE is useful for is a recording of Kaspersky Anti-Virus. Kaspersky Anti-Virus is commerical software that analyzes programs for probable malware and viruses in order to prevent malacious programs from running. However, Anti-Virus software often does interesting actions. After running STAVE on a systems trace containing Kaspersky Anti-Virus doing a normal execution, we have the following information (narrowed to 'avp.exe' for the sake of brevity, which we know is the anti virus processes name):

```
proc(3448,WmiPrvSE.exe) : 92
  proc(1404,avp.exe) : {[section, 19]}
proc(1404,avp.exe) : 55
  proc(3448,WmiPrvSE.exe) : {[section, 19]}
  proc(440,csrss.exe) : {[ALPC, 14][section, 2][virtualmemory, 8]}
  proc(2344,explorer.exe) : {[section, 7][virtualmemory, 2]}
  proc(588,lsass.exe) : {[section, 2]}
  proc(696,svchost.exe) : {[section, 1]}
proc(2448,avp.exe) : 39
  proc(2448,avp.exe) : {[virtualmemory, 15]}
  proc(496,csrss.exe) : {[ALPC, 4][section, 1]}
  proc(2344,explorer.exe) : {[virtualmemory, 6]}
  proc(3556,rundll32.exe) : {[section, 2][virtualmemory, 5]}
  proc(1180,svchost.exe) : {[file, 6]}
proc(440,csrss.exe) : 72
  proc(1404,avp.exe) : {[ALPC, 14][section, 2][virtualmemory, 5]}
proc(496,csrss.exe) : 138
  proc(2448,avp.exe) : {[ALPC, 4][section, 1]}
proc(2344,explorer.exe) : 816
  proc(1404,avp.exe) : {[section, 7][virtualmemory, 684]}
proc(588,lsass.exe) : 6
```

```
  proc(1404,avp.exe) : {[section, 2]}
proc(3556,rundll32.exe) : 122
  proc(2448,avp.exe) : {[section, 2]}
proc(696,svchost.exe) : 9
  proc(1404,avp.exe) : {[section, 1]}
proc(1180,svchost.exe) : 5
  proc(2448,avp.exe) : {[file, 5]}
```

One notable part of this is that 'avp.exe', uses two processes, and that one of those has a inflow of information from explorer.exe, also know as Windows Explorer, which allows users to view the file system. This makese some amount of sense, as the Kaspersky Anti-Virus's job is to scan files for malicious content. Although it isn't clear what exactly the avp.exe process is doing, it would certainly need file system access. Oddly though, it does not appear that the two avp.exe processes are communicating, at least in a observable way. Possibly however, they are communicating through a shared address space created when the processes forked from each other. Unfortunately, if they are not using a system call to do this it is hard for us to see, especially since the original process creation system call was before the start of this recording. To catch this, we would need to start the PANDA recording from boot up, when the anti virus software was started.

6.2 Encryption Extortion Malware

Another sample program trace that we would like to gain an understanding from is malware samples. In particular, Brendan Dolan-Gavitt has created huge warehouse of PANDA recordings for malware, called malrec[7]. One of these recordings is an example of encryption extortion malware, which encrypt the important files of a user, then demand a cryptocurrency payment in exchange for the decryption key, which has been sent to a server. This example malware is notable in that it claims it is from the FBI as a scare tactic[8]. Here's the output of STAVE for the malware recording:

```
proc(848,5026b6f33182fd) : 91
  proc(508,SearchProtocol) : {[section, 11]}
  proc(776,cmd.exe) : {[section, 1]}
  proc(364,csrss.exe) : {[ALPC, 4][virtualmemory, 1]}
  proc(1028,dllhost.exe) : {[section, 51]}
  proc(1720,explorer.exe) : {[section, 4]}
  proc(976,ipconfig.exe) : {[section, 14]}
  proc(828,svchost.exe) : {[section, 5]}
proc(508,SearchProtocol) : 53
  proc(848,5026b6f33182fd) : {[section, 10]}
proc(776,cmd.exe) : 205
  proc(848,5026b6f33182fd) : {[section, 1][virtualmemory, 3]}
proc(364,csrss.exe) : 232
  proc(848,5026b6f33182fd) : {[ALPC, 4][virtualmemory, 1]}
proc(1028,dllhost.exe) : 148
  proc(848,5026b6f33182fd) : {[section, 49]}
proc(1720,explorer.exe) : 107
  proc(848,5026b6f33182fd) : {[section, 4]}
proc(976,ipconfig.exe) : 124
  proc(848,5026b6f33182fd) : {[section, 15]}
proc(828,svchost.exe) : 107
  proc(848,5026b6f33182fd) : {[section, 5]}
```

In this trace, the malware is an autogenerated name, 5026b6f33182fd, and is executed via the command prompt, cmd.exe. Unfortunately, it is still difficult to tell what this malware is doing, though it seems to be using search and networking. This probably indicates that the malware is doing something sneaky to evade detection.

7 Future Work

There are several ways that that the data extraction of Systems Trace Analysis Visualization (Entire) could be extended. The first is to ensure that all ways processes can communicate are covered. Adding more supported operating systems would be another way to extend this system. Finally, applying this across other systems to track the flow of information at an abstract level could be done. Other forms of data analysis could also be applied, allowing finer grain approaches. We could segmet the communication flows by time, allowing users to approximate when communication is happening in the replay. We could also attempt to do some 'light weight' style analysis to try to associate flows into groupings and try to properly build out a tree of information flow.

7.1 More Mechanisms

Additional data collection mechanisms would be useful. Another form of light weight taint analysis would be to try to match the data flows across multiple known flows. For instance, if a file is read and then sent onwards to another process, it would be nice to associated those. Of course, this would miss any mangling of the data such as decryption. It would also miss partial flows, as well as falsely associate common or short data flows like null pointers. Additionally, collecting a rough count of how much information processes pass back and forth would be useful. For instance, if a process writes to a file, and then another reads that file, getting a sense of how much information each makes would be good.

7.2 Further Examples

Another necessary step for developing STAVE further is more complete unit examples. Unfortunately, trying to use this on malware and interesting examples where we don't know what's going on makes it really difficult to distinguish unknown behavior from diffencies in STAVE. Thus, having several examples of base truth that show example information flows would be useful, especially in the cases that are subtle and uncommon, such as section use with files.

7.3 Linux & other Operating Systems

Another useful area to expand STAVE would be to implement it in other operating systems. In particular, expanding to popular Linux distributions such as Debian or Ubuntu would allow a greatly increased user base. Unfortunately, the data flows do depend on the system calls in use, and doing so would be a lot of effort and reimplementation. Despite this, implementing a similar system in linux would be much easier as the linux operating system is open source and well documented.

7.4 Applications to other fields of study

This approach is also applicable to other areas, and could be used to gain insight into them. One application would be in computer networks, using IP addresses and ports to track what computers are talking, as well as looking at what protocols they are using.

8 Conclusion

Systems Trace Analysis Visualization (Entire) provides a cheap, easy to use visualization of inter-process communications in a PANDA replay. This allows it to quickly approximate areas in what would otherwise take a much longer time for heavier weight analyses. Of course, though this approach is fast it serves best as a starting point for further analysis, and it does not completely cover the realm of ways processes may communicate. Despite this, STAVE fills in a gap in the current PANDA based reverse engineering process, allowing faster reverse engineering 'for the greater good' as the PANDA paper explains. This tool will help the growing PANDA community understand their program traces, as well as help ease the learning curve of the PANDA tool.

References

[1] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, R. Whelan. Repeatable Reverse Engineering for the Greater Good with PANDA. TR CUCS-023-14

[2] R. Whelan, T. Leek, D. Kaeli. Architecture-Independent Dynamic Information Flow Tracking. 22nd International Conference on Compiler Construction (CC), Rome, Italy, March 2013.

[3] https://msdn.microsoft.com/en-us/library/windows/hardware/ff567122(v=vs.85).aspx

[4] https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx

[5] https://github.com/volatilityfoundation/volatility

[6] https://github.com/moyix/panda/blob/master/docs/pandalog.md

[7] https://github.com/moyix/panda-malrec

[8] http://www.fbi.gov/news/stories/2012/august/new-internet-scam/new-internet-scam